# Makefiles

# From the Manual Page on Make

- The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them.
- You can use it to describe any task where some files must be updated automatically from others whenever the others change.
- To prepare to use make, you must write a file called the makefile that describes the relationships (dependencies) among files in your program, and the states the commands for updating each file.
- In a program, typically the executable file is updated from object files, which are in turn made by compiling source files.
- Once a suitable makefile exists, each time you change some source files, this simple shell command:

```
make
```

suffices to perform all necessary recompilations. The make program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

# Dependency Diagram

$$\begin{tikzcd}
 & & \text{XXXXXX modified} & & \\[-2em]
\boxed{\text{main1.c}} \arrow[d, swap, "\text{compile}"] & & \boxed{\text{defs.h}} \arrow[dll] \arrow[drr] & & \boxed{\text{func.c}} \arrow[d, "\text{compile}"] \\
\boxed{\text{main1.o}} \arrow[drr] & & & & \boxed{\text{func.o}} \arrow[dll, swap, "\text{linking}"] \\
 & & \boxed{\text{prog1.exe}} & &
\end{tikzcd}$$

## Makefile Format

- Makefile contains (i) Definitions (ii) rule statements
- Rule Statements have the following format:

```
target1:        dependencies
                action1
                action2
                ...

target2:        dependencies
                action1
                action2
                ...

...
...
```

```
main1.o:     main1.c defs.h
             gcc -c main1.c
```

At least one tab before dependencies and actions (as seperator character)

## Example 1

```
prog1.exe:   main1.o func.o
             gcc main1.o func.o -o prog1.exe
             echo "done”

main1.o:     main1.c defs.h
             gcc -c  main1.c

func.o:      func.c defs.h
             gcc -c func.c
```

# Example 2

```
all:            prog1.exe prog2.exe


prog2.exe:      main2.o func.o
                 gcc main2.o func.o -o prog2.exe -lm

prog1.exe:      main1.o func.o
                 gcc main1.o func.o -o prog1.exe -lm

main1.o:         main1.c defs.h
                 gcc -c  main1.c

main2.o:         main2.c defs.h
                 gcc -c  main2.c

func.o:         func.c defs.h
                 gcc -c func.c
```

## Some notes

- By default, make builds the first target (the dependencies are built as needed).
- In order to build the others (other than the first), you can give the name of the target on the command line:

```
make target
```

- You can also add a dummy node (all target) as the first target to build multiple targets (see Example 3).

- By default, make reads the files named makefile or Makefile. In order to make it process some other file (e.g. mymakefile), use the –f option:

```
make –f mymakefile
```

# Example 3

```
INCDIR =        /home/canozturan/CMPE230/MAKEEXAMPLES/1/INCLUDE

all:            prog1.exe prog2.exe


prog1.exe:      main1.o func.o
                gcc main1.o func.o -o prog1.exe -lm
                echo "done"

prog2.exe:      main2.o func.o
                gcc main2.o func.o -o prog2.exe
                echo "done"

main1.o:        main1.c $(INCDIR)/defs.h
                gcc -c  -I $(INCDIR)  main1.c

main2.o:        main2.c $(INCDIR)/defs.h
                gcc -c  -I $(INCDIR) main2.c

func.o:         func.c $(INCDIR)/defs.h
                gcc -c -I $(INCDIR) func.c
```

# For more details on Make

- Have a look at the 212 page  Gnumake manual

  https://www.gnu.org/software/make/manual/